KATSU
DEVELOPER MANUAL

Version Beta 1.0

Table of Contents

1 Introduction

Katsu is a light cross-platform C library for 2D video game programming. It includes an API for 2D rendering, a simple input system, and sound output. Tile renderers of old school computers and video game consoles are the basis for Katsu's 2D rendering capabilities. The kind of images that Katsu can produce outperforms most of these systems; this is to allow programmers some freedom when designing games. With this in mind, some features present in old systems go against modern GPU hardware and graphics APIs, which is why they are not present (like some per horizontal line effects).

1.1 Initial steps

Firstly include **katsu/kt.h** to all source files that use the library. To initialize Katsu you call `kt_Init()`, this will initialize all of the subsystems that it uses (Video, Joypad, Graphics and Audio), if an error occurs then the return value will be non-zero. After Katsu is initialized an *operating system window* (OSW) will be created and shown on screen. To end all systems, the user can call `kt_Exit()` where a status code can be passed (you can freely define the meaning of each code). After Katsu is initialized an *operating system window* (OSW) will be created and shown on screen.

1.1.1 Main application loop

A normal Katsu application should do the following things in order: read input, update the state of the application, draw to the screen. It is recommended that the application loops forever until the end user or the application decides to exit, this is called the *main application loop*. To read the input use the `kt_Poll()` function (this will also handle events received by the OSW). To draw to the screen use the `kt_Draw()` function. The following code snippet is a simple example of this:

Code 1.1

```c
#include <katsu/kt.h>
int main()
{
    /* Initialize Katsu */
    if (kt_Init()) {
            return 0;
    }
    /* Initialize application state */
    /* Main application loop */
    while(1) {
            kt_Poll(); /* Handle and read inputs */
            /* Update application state */
            kt_Draw(); /* Draw to the screen */
    }
}
```

1.1.2 Custom defined exit function

Since an infinite loop is used, a close event of the OSW will exit the application (same as calling `kt_Exit()`). It is common to ask if it is fine for the application to exit or do other background work before exiting (saving the application state to an external file, for example). You can use `kt_ExitFuncSet()` to pass a user defined function that Katsu will call before exiting the application. This function will receive the status code passed to `kt_Exit()` and returns a code that can cancel `kt_Exit()` from running. This behavior is shown in the example below:

Code 1.2

```
#include <katsu/kt.h>
u32 askToClose(u32 status)
{
    /* Ask user for confirmation on progam exit via a loop */
    if (/* Wants application to exit*/) {
            /* Save application state */
            return KT_EXIT;
    } else { /* Wants to remain */
            return KT_EXIT_CANCEL;
    }
}

int main()
{
    /* Initialize Katsu and application state */
    kt_ExitFuncSet(askUserClose);
    /* Main program loop */
}
```

1.2 Predifined primitive types

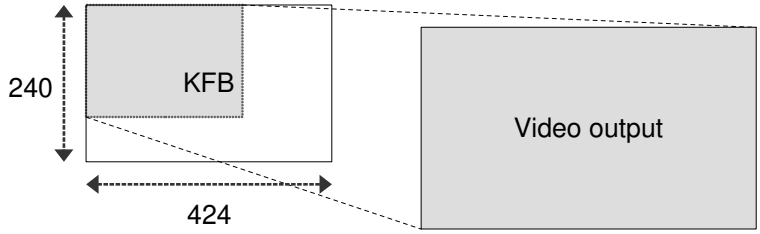
TO DO

2 Video

As mentioned, Katsu presents an OSW when initialized. The Video subsystem can control some aspects of the OSW and how the drawn image is presented inside its frame. One can programatically change the OSWs *frame* with preset sizes by using `kt_VideoFrameSet()`, for example, passing `KT_VIDEO_FRAME_2X` shows the frame at exactly 2 times the size, when `KT_VIDEO_FRAME_FULLSCREEN` is passed, the OSW will cover the television/monitor screen (to exit full screen mode the user can press the *Esc* key on the keyboard). Finally, the frame's title can be set with `kt_VideoTitleSet()`.

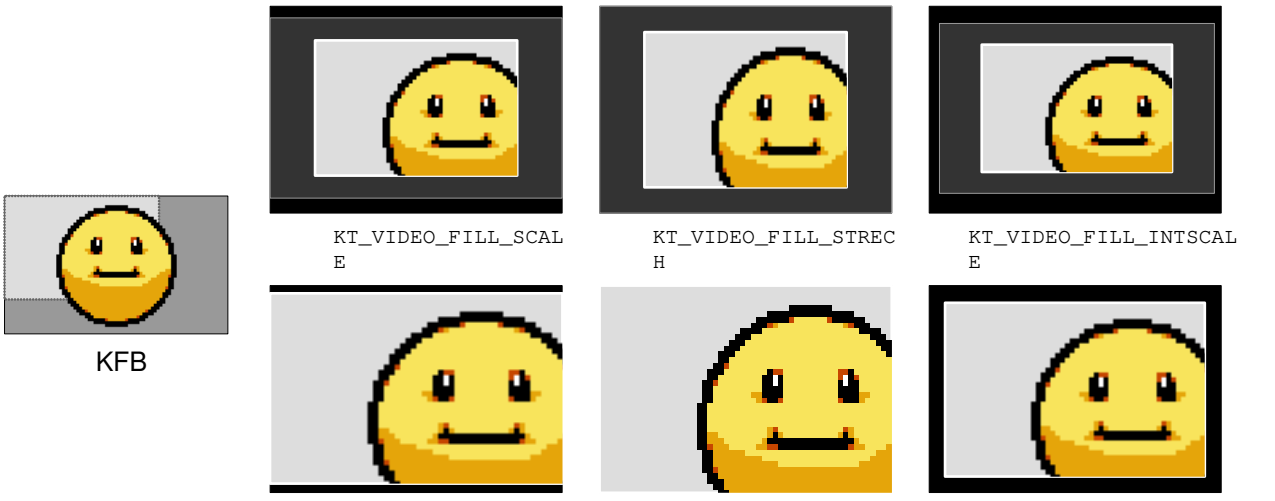
2.1 Defining the video output

Standard Katsu will always draw a 424x240 pixel image called the *Katsu frame buffer* (KFB). By default, the whole image will be shown on the OSW, with `kt_VideoOutputSet()` you can crop the KFB to only show a certain width and height in pixels (the minimum values for both are 32 pixels), this is known as the *video output*. This way, you can easily change between different video resolutions.



2.2 Fill modes

Most likely the video output will be smaller than the frame of the OSW. With `kt_VideoFillModeSet()` you can specify the way the video output fills the OSW. The following figure shows how all the preset options act:



The normal `KT_VIDEO_FILL_*` options center the video output inside the KFB and uses its dimensions to fill the screen, pixels outside the video output are not displayed. For `KT_VIDEO_FILL_OUT_*` options the dimentions used are those of the video output.

2.3 Filters

TO DO

3 Joypad

A *joypad* is the standard device for controlling the application, it supports 14 digital buttons and two analogue sticks. A joypad will connect to one of an expected maximum of 4 *ports*, this is an artificial maximum and can be changed in **katsu**/**joypads.h** when compiling the library. Since Katsu has no real controller, the button layout can be set to be whatever you want. The suggested use is the following:

Before reading the state of the joypad you must first poll them with `kt_Poll()`, this function is also used to listen to other events, so this should be made at the start of the main application loop. The `kt_JoyIsActive()` function returns a non-zero value if there is a controller connected at that port. By using the `kt_JoyButton*()` functions you can retrieve the state of the buttons in three conditions (whether they are held in, just been released or just been pressed), pressed buttons can be tested by doing following:

```
kt_Poll(); /* Poll the joypad state */
/* Check if A and START buttons in joypad 0 have been pressed at the same time */
if (kt_JoyButtonDown() & (JOY_A | JOY_STR)) {
    /* Do something */
}
```

The state of the joysticks can be read by using `kt_JoyStick()` with the stick ID being an axis of the right or left stick. The value of the axis will be returned as a signed 8-bit value (the values will go from -128 to 127).

4 Graphics

4.1 Rendering model

TO DO

4.1.1 Back Color

TO DO

4.1.1 Offset Color

TO DO

4.1 Graphics Formats

TO DO

4.1.1 Tiles

A *tile* refers to a single 8x8 pixel image. Katsu supports tiles that are 4 bits per pixel (4bpp), which is a paletted format where each pixel references one of 16 colors. As such, we need 32 contiguous bytes to store a tile:

0	0	8	8	9	9	0	0
0	8	9	A	C	B	9	0
8	8	A	A	C	D	B	9
7	8	A	A	A	B	B	9
7	8	8	9	A	A	9	8
1	9	8	8	9	9	8	7
0	1	9	9	9	8	7	0
0	0	1	1	1	1	0	0

Each byte holds two pixels, the lower 4 bits correspond to the leftmost pixel, and the higher 4 bits correspond to the rightmost pixel. Since each pixel holds an index to a 16 entry color palette, the tiles color depends on the current state of Color Memory (CMEM). Katsu's Tile Memory can store up to 16,384 tiles at once, each tile has a corresponding ID assigned, which is a 14 bit value. You can load a set of contiguous tiles to the Tile Memory by using `kt_TilesetLoad()` starting from an initial tile ID. When loading, the tiles are copied to Tile Memory so changes to loaded tile graphics must be updated by loading them again.

4.1.2 Tilemaps and Characters

A *tilemap* is composed of 64x64 *characters*, where the first 64 characters refer to the first row of the tilemap, there are 16 tilemap. One character is 4 bytes long and stores information in the following order:

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
pal	_padding	fthi	tlo

Where `pal` refers to the number of the palette used for the tile. The `fthi` byte stores the following information:

Bits	Name	Description
0-5	thi	Highest 5 bits of the tile ID.
6	hf	Flips the tile horizontally when set.
7	vf	Flips the tile vertically when set.

The value of `thi` (highest 5 bits) and `tlo` (lowest 8 bits) are joined to produce the 14 bit tile ID. A single character of a tilemap can be set by using `kt_TilemapSetChr()`, for loading a larger amount of tilemap data at once use `kt_TilemapLoad()`, this function allows for loading a sub-rectangle of a larger map to the Tilemap Memory.

4.1.3 Colors

Individual *colors* are stored in RGBA format, where each component is 8 bits. The last byte of a color (also called the alpha component) is ignored and only there to have colors 32-bit aligned:

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
red	green	blue	alpha

4.1 Sprites

Sprites are individual images made out of joined tiles that can move around the screen independently from each other. There is a hard limit of 2048 sprites drawn in a single pass for all layers (this means, for example, that if one layer draws 2047 sprites, then another layer can only draw 1 more sprite). Each sprite has it's own individual attributes which are specified in 16 bytes divided across four 32-bit values:

4.1.1 POS

1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10	F E D C B A 9 8 7 6 5 4 3 2 1 0
pos_y	pos_x

The POS value specifies where in the screen will the sprite be drawn, the coordinates dictate the top-left corner of the sprite, the coordinates are signed values and go from -32768 to 32767, this value can be constructed with the `KT_SPR_POS()` macro:

Bits	Name	Description
0-15	pos_x	Y coordinate of the sprite.
16-31	pos_y	X coordinate of the sprite.

4.1.2 CHR

1F 1E 1D 1C 1B 1A 19 18	17 16 15 14	13 12 11 10	F	E	D C B A 9 8 7 6 5 4 3 2 1 0

pal	vsize	hsize	vf	hf	tid

The CHR value specifies the set of tiles that will be drawn, the palette used and if any mirroring is to be applied, this value can be constructed with the `KT_SPR_CHR()` macro:

Bits	Name	Description
0-13	tid	The tile ID of the top-left corner.
14	hf	Horizontal flipping
15	vf	Vertical flipping
16-19	hsize	Width of sprite (mult of 8).
20-23	vsize	Height of sprite (mult of 8).
24-31	pal	Palette number used.

4.1.3 SFX

1F	1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10	F E D C B A 9 8	7 6 5 4 3 2 1 0
bl_act	hue	hue_alpha	alpha

The SFX value specifies special effects that are to be applied to the sprite: hue mixing and transparency. This value can be constructed by OR'ing the `KT_SPR_HUE()` and `KT_SPR_BLEND()` macros:

Bits	Name	Description
0-7	alpha	The alpha value of the sprite.
8-15	hue_alpha	Amount of hue mixing
16-30	hue	Hue to be mixed to the sprite
31	bl_act	Transparency active toggle.

4.1.4 MAT

1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8	7 6 5 4 3 2 1 0
unused	mat_idx

The MAT value specifies the index of the matrix loaded in Matrix Memory that will be applied to the sprite, the index 0 refers to the identity matrix, this matrix is the only constant matrix in Matrix Memory. This value can be constructed by using the `KT_SPR_MTX()` macro:

Bits	Name	Description
0-7	mat_idx	Index of the matrix to apply.

4.1.5 Matrix Memory

TO DO

4.1 Layers

TO DO

4.1.1 Normal Map Layers

TO DO

4.1.1 Scroll Map Layers

TO DO

4.1.1 Affine Map Layers

TO DO

4.1.1 Sprite Layers

TO DO

4.4 Blending

TO DO

4.4 Windows

TO DO

4 AUDIO

TO DO